On Model-Driven Engineering to implement a Component Assembly Compiler for High Performance Computing

Julien Bigot — Christian Pérez

LIP, ENS Lyon
46 allée d'Italie
69364 Lyon Cedex 07, France
{julien.bigot,christian.perez}@inria.fr

ABSTRACT. *High performance scientific applications provide very interesting challenges from the software engineering point of view. In addition to their high performance requirement, their codes exhibit a long life cycle that includes reuse as part of code-coupling applications. Dedicated programming models are required to ease the adaptation of these codes over time without introducing overhead at runtime. HLCM is a component assembly model that supports high level concepts easing code adaptation. It prevents runtime overheads by implementing these concepts through transformation applied at deployment to generate a concrete assembly. This paper deals with our experience with metamodeling and model transformation as used to implement HLCM. It provides some feedback on the advantages and drawbacks we found by using a model driven approach.*

RÉSUMÉ. *Les applications scientifiques haute performance présentent des défis très intéressants du point de vue du génie logiciel. En plus de leur besoin de performances, leurs codes possèdent un cycle de vie long qui comporte des réutilisations au sein d'applications de couplage de codes. Ainsi, des modèles de programmation spécialisés sont requis pour faciliter l'adaptation de ces codes au cours du temps sans introduire de sur-coûts à l'exécution. HLCM est un modèle d'assemblage de composants qui comporte des concepts de haut niveau facilitant l'adaptation de codes. Il évite les sur-coûts à l'exécution en mettant en œuvre ces concepts par une transformation appliquée lors du déploiement pour générer un assemblage concret. Ce papier s'intéresse à notre expérience avec la métamodélisation et la transformation de modèles tel qu'utilisées pour mettre en œuvre HLCM. Il offre des retours sur les avantages et inconvénients que nous avons trouvés à utiliser une approche basée sur les modèles.*

KEYWORDS: *Software Components, High-Performance Computing, Model Transformation*

MOTS-CLÉS : *Composants logiciels, Calcul de haute performance, Transformation de modèle*

1. Introduction

High-performance computing is a field of computer science that provides very interesting challenges from the software engineering point of view. It combines two conflicting requirements: high performance and long life-cycle of codes ($\sim$30 years) compared to machines ($\sim$3 years). To deliver the expected performance, applications are executed on dedicated hardware such as supercomputers and fine-tuned for it. However, the long life-cycle of applications requires to adapt them to new hardware. Doing so manually means high cost.

To support these specificities, it is important to offer scientists, who are not expert in (parallel) programming, a suitable programming model. Such a model should introduce minimal overhead and support parallelism. It should ease the adaptation of applications to various hardware architectures by allowing the use of algorithms optimized for a given hardware and/or communication model. It should also ease the reuse of existing pieces of code in different context.

An interesting model is offered by software components [SZY 02]. Typically, components are developed using an external paradigm (*e.g.* object oriented) and they expose a set of ports describing used and provided services, typed by object interfaces. The component instantiation and connexions are handled via a dedicated API or a language (ADL). Components ease code-reuse thanks to their clearly identified points of interaction. Components also ease optimization of applications: only the assembly has to be modified in order to replace a component by another one optimized for a specific hardware. Some component models such as the CORBA Component Model or EJB support network transparency by enforcing a remote method invocation mechanism for inter-component interactions. This network transparency does however introduces too much overheads for invocations between components in the same process. Component models dedicated to high performance have thus been developed such as the Common Component Architecture (CCA). These models do however tend to offer a low level of abstraction. For example, CCA does neither handle parallel communications nor a standard model to optimize an application for various platforms.

This paper deals with the High Level Component Model (HLCM), a model aiming to solve these issues and whose implementation is based on MDE. The remaining of the paper is organized as follow. Section 2 presents an overview of HLCM, while Section 3 describes the involved metamodels and model transformation. Then, Section 4 provides some feedback regarding this experience with (meta)modeling and Section 5 concludes the paper.

2. Overview of High Level Component Model

HLCM [BIG 10] is a hierarchical and generic component model with connectors originally designed for high performance applications. The principle of HLCM is to rely on an underlying model that supports multiple kinds of interactions and to generate an assembly optimized for the available hardware when an application is deployed.

For example, network support for an interaction between two component instances can be generated only if necessary. It also enables to choose the best suited implementation of a given algorithm amongst those available.

HLCM can be seen as a component assembly compiler. It exposes to the user an assembly model with a high level of abstraction and generates a concrete assembly optimized for a given hardware architecture. There are three main concepts supported in the high level assembly model: **connectors** to support identification of the codes responsible for inter-component interactions as opposed to application logic implementation; **multiple implementations** for component and connectors to support code adaptation to the platform where it is deployed; **generic programming** to support identification of reusable composition patterns similarly to algorithmic skeletons and to let some aspects of applications vary (e.g, the number of processes to use).

In order to implement the transformation implied by HLCM, we chose to follow a model based approach. The assemblies of components are models of applications, and the models of these models (*i.e.* metamodels) have been described. The transformation is very similar to the Model Driven Architecture (MDA) as advocated by the OMG. A source model (PIM in MDA) is transformed to a platform-specific model (PSM) based on a description of the platform (PDM). What we define as the platform are the hardware resources where the application is deployed and their connexions.

HLCM does not fix the underlying component model used at execution. Therefore, some aspects of the metamodels depend on this choice. Moreover, HLCM is independent of the way the implementations are chose and the platform is described. This means that what HLCM offers is a metamodeling framework that can be specialized for various underlying component models rather full metamodels. Specializations of HLCM have been implemented for CCM and minimal component models supporting `Java` and `C++` components with local, MPI and CORBA interactions [BIG 10].

3. Modeling HLCM

HLCM has been implemented using the tools provided as part of the Eclipse Modeling Framework (EMF). The metamodels for the source and destination assemblies have been described using ECORE (171 meta-classes). The concrete syntax for the source model has been described using XTEXT. The model transformation has been implemented in plain JAVA due to the lack of mature dedicated tools at the time.

The three main concepts of the source model are components, port-types and connectors. A component describes a unit of computation and a connector a kind of inter-component interaction. Connectors expose a set of roles that are fulfilled by ports. The destination model describes a concrete instance of the application; its main elements are component instances, ports and connexions (connector instances).

During the modeling of HLCM, some modeling patterns have been conceived and applied. A first pattern has been identified to model multiple implementations of both

components and connectors. It mainly consists in distinguishing three elements in the modeling: the **type** itself, the (possibly multiple) **implementations** of the type that reference the type they implement, and the **instance descriptors** that reference a type and in the case of the concrete destination model, an implementation. The class used for modeling implementations is usually abstract as HLCM supports distinct kinds of implementations (*e.g.* composite implemented by an assembly *vs.* primitive supported by the underlying model).

The second pattern has been conceived to model genericity for components, port types and connectors similarly to `C++` templates. An abstract class modeling generic parameters is created and referenced by types to make them generic. This class is specialized for each kind of element that can be used as generic parameter, to be used instead of the element itself (*e.g.* a `ComponentParameter` can be used where a `Component` is expected). Instance descriptors are extended with a reference to arguments that associate each parameter to a value. The class modeling arguments is abstract and specialized just like the one modeling parameters.

To support multiple models at runtime, the classes used to model primitive implementations are abstract. They have to be extended for each target model. In the destination model however, HLCM instances reference an implementation from the source model; there is no need to make any modification to this model.

The transformation from the source to the destination model is a rather simple algorithm applied until convergence. The implementations of the elements of the application are chosen according to an externally provided heuristic. The elements whose implementation is composite (*i.e.* an assembly) are replaced by this assembly. If elements inside this assembly are typed by a generic parameter, the type is replaced by its value obtained from generic arguments in the current context. The transformation is finished when all element types are primitive.

4. Feedback

From this experience with modeling, the first remark that comes to mind is the high speed of development made possible by this approach together with its associated tools. A first attempt had been made to implement HLCM with plain JAVA but it was abandoned after about one month since it has been possible to reach roughly the same state in about two days using ECORE. Tools that enable to associate a concrete syntax with a model are also really time savers.

Understanding the concepts behind modeling (*e.g.* association *vs.* containment) is quite intuitive for someone familiar to UML. However, a confusing aspect appears with the various levels of modeling (*i.e.* model *vs.* metamodel) as it seems that some elements can be put at various levels. In HLCM for example, the various types of interactions are first class elements (namely connectors) while there are usually only a fixed set of interactions supported in classical component models. This means that the usual approach consisting in describing a set of meta-classes in the metamodel

to model these interactions cannot be used. Instead the types of interactions must be described by an element of the model and this difference of modeling level must be kept in mind when implementing the transformation. This is even more apparent with generic programming that let higher level elements be manipulated in the model.

Regarding the model transformation, its main complexity comes from the interface to access the models from JAVA. The EMF tools provide a rather intuitive access to the model by generating getter and setters and by respecting the JAVA `Collection` interface for non unique multiplicity. This does however mean that accessing an element of such a collection given an identifying key (e.g., its name) requires to go across the whole collection. This makes the code somehow complex to understand and is not very good performance-wise. Another source of complexity stems from the omnipresent type casts required because some constraints on the model that can not be expressed in ECORE. *e.g.*, the value of an argument associated to a `ComponentParameter` is a `Component`, but this constraint can not be expressed in the model itself.

These two limitations could be solved by using a domain specific language for transformation such as QVT or ATL and a language to express additional constraint on the model such as OCL. In our use-case, it is very important to allow the transformation described in a dedicated language to be interfaced with native code (e.g., JAVA) in which the heuristics of choice will very likely be implemented. At the time where the model transformation was implemented however there was no mature enough implementations of these languages hence our choice of plain JAVA.

5. Conclusion

This paper has presented HLCM, a component assembly model that supports hierarchy, connectors, multiple implementations, and generic programming. The use of an MDE approach has made possible a really quick implementation of the whole transformation from source model in textual form to the executable destination model. Some difficulties that are mainly due to the lack of maturity of the tools have been identified.

The HLCM assembly model is static. Once transformed, it remains the same for the whole execution of the application. However, a dynamic behavior is required for self-adaptation (e.g., load balancing) and/or for workflows. Therefore, we plan to explore how the field of models@runtime could help us to support dynamicity in HLCM.

6. References

[BIG 10] BIGOT J., "Du support générique d'opérateurs de composition dans les modèles de composants logiciels, application au calcul à haute performance", PhD thesis, INSA de Rennes, Dec. 2010.

[SZY 02] SZYPERSKI C., GRUNTZ D., MURER S., *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 2 edition, 2002.